LA-UR-02-0852

Approved for public release;
distribution is unlimited.

Title:	MATLAB-Based VHDL Development Environment
Author(s):	Kimberly K. Katko kkatko@lanl.gov, (505) 665-5134 and Scott H. Robinson shr@lanl.gov, (505) 665-1954 MS D448, PO Box 1663 Los Alamos National Laboratory Los Alamos, NM 87545 Fax: (505) 665-4657
Submitted to:	Engineering of Reconfigurable Systems and Algorithms June 24-27, 2002, Las Vegas, Nevada, USA

MATLAB-Based VHDL Development Environment

Kimberly K. Katko
MS D448, P.O. Box 1663
Los Alamos National Laboratory
Los Alamos, NM 87545

Scott H. Robinson
MS D448, P.O. Box 1663
Los Alamos National Laboratory
Los Alamos, NM 87545

Abstract

The Reconfigurable Computing program at Los Alamos National Laboratory (LANL) required synthesizable VHDL Fast Fourier Transform (FFT) designs that could be quickly implemented into FPGA-based high speed Digital Signal Processing architectures. Several different FFTs were needed for the different systems. As a result, the MATLAB-Based VHDL Development Environment was developed so that with a small amount of work and forethought, arbitrarily sized FFTs with different bit-width parameters could be produced quickly from one VHDL generating algorithm. The result is highly readable VHDL that can be modified quickly via the generating function to adapt to new algorithmic requirements. Several additional capabilities are integrated into the development environment. These capabilities include a bit-true parameterized mathematical model, fixed-point design validation, test vector generation, VHDL design verification, and chip resource use estimation.

1 Introduction

LANL needed the flexibility to build a wide variety of FFTs with a quick turn around time. It was important to have an effective way of trading off size, speed and precision. The FFTs also needed to be efficiently implemented into our existing FPGA-based architecture. Reconfigurable computing systems at LANL have been designed to accept two or four inputs on each clock. This allows the data processing rate to be reduced to a more manageable speed. This approach, however, limits us from using existing FFT cores.

A MATLAB-Based VHDL Development Environment (MBVDE) was created in response to our FFT needs. MBVDE provides more flexibility than is available with VHDL. The technique allows new designs to be implemented and verified quickly. In addition, analysis tools are incorporated to evaluate trade-offs.

MBVDE incorporates the performance of VHDL, the fast design time of core generation, and the benefit of not having to know VHDL available with C-tools into one environment. The MBVDE approach is not a comprehensive solution, but is a powerful method for algorithms that involve the cascading of fundamental building blocks.

2 VHDL Development Environment Comparisons

There are advantages and disadvantages of the different development tools. The trade-offs include development time required, flexibility of design, control over design details and overall performance, code readability and visibility, and the need to know VHDL. These trade-offs are considered as a function of different tools in this section. Tools contrasted include cores, high-level tools, hand coded VHDL, and LANL's MBVDE. Commercially or publicly available IP cores and vendor supplied cores or core generators will be grouped into one category and referred to as cores. High-level tools such as C-to-VHDL and C-to-EDIF tools will also be grouped into one category and referred to as high-level tools.

2.1 Development Time

One of the most important considerations when porting an algorithm to an FPGA is development time. Of the three standard methods, development time using cores is the fastest. Using high-level tool takes significantly longer, while digital design using VHDL takes longer still. For our design method, the initial development of an algorithm for automated VHDL generation will take slightly longer than that of standard hand coding of VHDL. Once this groundwork is laid, the generation of new VHDL with the MBVDE can be done with speeds comparable to those using core generation programs.

2.2 Flexibility

Another important aspect of any design method is the inherent flexibility afforded to the designer. In this area, cores can be particularly poor, as the options are severely limited. This lack of flexibility limits the applications for which this core can be used. High-level tools and hand coding provide more flexibility, but only within the individual application being written. If one decided to change the length of an FFT, for example, new code and new parameters must be provided. With MBVDE, an unlimited number of input parameters can be built into the design. The result is tremendous flexibility within the chosen algorithm, where the designer can effectively decide how much of the problem should be constrained. The downside is that not all functions are good candidates for this type of design. The types of designs that are best suited to the MBVDE are ones based on algorithms that have regularly repeating (cascaded) structures. FFTs, lattice filters, recursions such as Levinson's and filter implementations such as polyphase structures and Hogenauer decimators are all examples of algorithms where cascaded versions of a fundamental base algorithm combine to form a more complex structure [1], [2], [3].

2.3 Design Control and Performance

Control over design details and subsequent design performance are always important. The highest performance designs will come from cores. When designing with high-level tools, details of the design are difficult to control. Unlike with cores, this lack of control is not a feature that aids in performance. Rather, tweaking the design to gain added performance is very difficult. When designing with VHDL, whether for a single hand-coded piece of code or for MBVDE, control over design details is straightforward.

2.4 Code Visibility and Readability

The cost of the high performance attained with a core is the lack of knowledge of design details. Visibility into the design provides the knowledge that is required to develop bit-true mathematical models and subsequent system verification tests. When the core is used in relative isolation this is not a big problem, because cores are generally tested thoroughly before they are released to the public. Lack of a bit-true model of the core does become a problem, however, when the core needs to be tightly integrated into a larger system. Designs formed using high-level tools often have similar limitations. Knowledge of the inner workings of the design is limited. In addition, these tools generally produce VHDL code that is difficult to follow and understand, even for experienced programmers. Code produced by tools that skip the VHDL step and go straight to producing EDIFs can be even more difficult to understand. This lack of code visibility contributes to a poor debugging environment and in extreme situations makes debugging impossible in the standard sense. VHDL, whether hand coded or produced as output from MBVDE, is straightforward to follow and understand assuming the code is clearly written and appropriately documented. This readability can be invaluable during the debugging process. Knowledge of the internal design details is also complete and it is therefore possible to build bit-true mathematical models to aid in the testing and debugging phases.

2.5 Knowledge of VHDL

When VHDL resources are limited, the need for the designer to have advanced VHDL coding skills also becomes an issue. There is no need to know VHDL to implement cores targeted for isolated modules. Some basic VHDL knowledge is required to integrate them with additional designs, however. While VHDL knowledge is not required to design with high-level tools, producing high-performance designs without it is difficult. Of course designing hand-coded VHDL or MBVDE generators requires knowledge of the language. The use of existing MBVDE generators, however, does not.

3 MBVDE Fundamentals

As discussed above, the LANL Reconfigurable Computing program required FFTs of varying size and precision for multiple projects. The desired FFTs needed to handle at least 100 Msamples/sec taking either 2 or 4 inputs per clock. Due to certain environmental requirements on memory usage, they could not

make use of select RAM. Trade offs between chip resources and numerical precision were critical to the designs. The initial algorithmic delivery required a 32-point FFT, but future applications had the potential to utilize sizes from 16 up to 1024 points. No existing FFT cores met these design constraints. Hand coding VHDL was the initial approach, but doing so for larger FFTs became increasingly tedious and accident-prone. At this point, the decision was made to leverage our knowledge of the basic repeating structures that make up an FFT to add some automation to the development process where appropriate. The MBVDE was developed in response to the above needs.

Automatic VHDL generation for specific applications has been done on several programs in the past [4], [5], [6], [7]. The unique feature of MBVDE is the additional capabilities that are built into the development environment. The capabilities include a bit-true parameterized mathematical model, fixed-point design validation, test vector generation, generation of synthesizable VHDL, VHDL design verification, and chip resource use estimation. The features, development and benefits of the LANL MBVDE FFT are discussed in the remainder of this section, using the FFT algorithm as a concrete example.

3.1 MATLAB as Development Environment

MATLAB was chosen as the development platform for several reasons. First, it is a widely used program that runs on many platforms. MATLAB is a natural fit for signal processing applications due to the built-in functions and available signal processing toolboxes and data visualization capabilities. The built-in FFT function made design verification very clear-cut. The fact that MATLAB is an interpreted language makes data manipulation and analysis quick and easy. A final reason for choosing MATLAB was that it provided an analysis environment for the easy creation of test vectors and the comparison of results.

3.2 Algorithmic Considerations

As previously described, this is not a comprehensive method applicable to every design situation. It is a powerful tool when cascading and placing smaller, fundamental building blocks creates the desired algorithm. Thus, before implementing an algorithm using the MBVDE, the desired algorithm itself must be analyzed for fitness to this technique and the component building blocks need to be identified.

3.2.1 Base design

For any cascaded or recursive style algorithm, a fundamental base algorithm is needed to eventually form the overall complex design. To enable this type of situation, LANL started with a hardware-verified, non-parameterized 32-point FFT. This base design implements a Decimation-In-Frequency Fast Fourier Transform (DIF-FFT) algorithm utilizing a heavily pipelined architecture to maximize data throughput. Inputs to the design are two real data samples. Output is one complex sample per clock cycle. The pipelined architecture allows data to enter and results to exit the logic on every clock cycle.

The DIF-FFT is built around a cascade of standard radix-2 butterfly computation modules [1]. This radix-2 module is the fundamental base algorithm for this particular design. Inputs to the radix are two complex data values and one complex twiddle value. The output is two complex result values. Each radix performs four complex additions, two complex multiplies and one complex vector multiply to correctly scale the output. The pipeline architecture of the radix allows new data to enter on each clock cycle and has a three-clock latency.

3.2.2 Identifying repeating structures

As stated previously, the primary use of this technique is for creating complex algorithms from smaller, fundamental building blocks. While the radix-2 is the primary building block, others are required to build a working larger FFT. The basic structures that make up an *N*-point FFT are shown below in Figure 1. For the current application, two points arrive at the input of the FFT on each clock cycle. The points arrive in natural order and need to be reordered (a bit reversing operation) before going into the first pass radix-2 components. Therefore, all *N* points must arrive before the input data reordering can begin. Once all points have arrived and the

reordering is complete, the data can be run through the first pass radix-2 components. Explicitly instantiating each of the radix-2 components allows the rest of the design to flow smoothly. All *N*/2 first pass radix-2 operations are completed simultaneously. The first pass radix-2 outputs then go through another round of reordering before being input to the second pass radix-2s. This procedure is followed for the remainder of the passes.

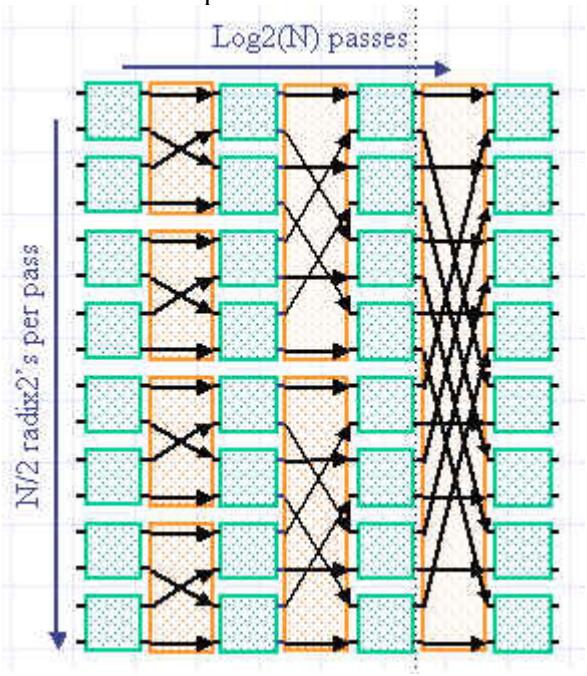

Figure 1

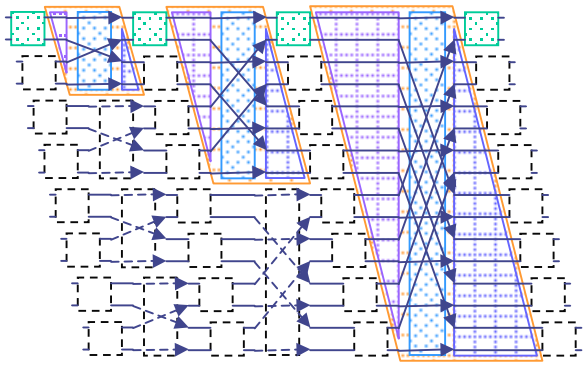

Figure 2

A space saving improvement can be made at this point, however. By instantiating just one radix-2 component per pass, the data for that pass can be run through the radix-2 sequentially, one pair of points at a time. Instantiating a single radix-2 component per pass requires some additional data management, but saves significant chip space, specifically in look up tables (LUTs). Holding the data points before feeding sequential pairs to the radix-2 requires the addition of a delay component. Following the radix-2 operation, the data points need to be held until enough are present to do the data reordering for that pass. This process of holding, reordering, delaying and running through the radix-2 is repeated for each pass. This new structure is shown in Figure 2. The structure of the FFT has now been broken down into its basic elements.

3.3 MATLAB Model Development Process

Once the mathematical algorithm has been analyzed on paper and broken down into the appropriate building blocks and a fundamental base algorithm, the process moves into the MATLAB environment so that models can be created and designs verified. It is in this stage that fixed-point models are created and the overall design correctness is verified by comparing to desired results with carefully chosen test vectors.

3.3.1 Identifying algorithm input parameters

Since the goal of this environment was to create FFT designs meant to accommodate several different algorithmic applications, the next step was to identify necessary input parameters so that all desired combinations were covered. The parameter choices for the varying FFT sizes were clear at this stage in the design: FFT size, input data bit-width, bit-widths of data after each radix-2 pass (internal scaling strategy), and twiddle factor bit-width were all important variables to allow use of the resulting FFTs in the different situations described previously. Writing code so that these parameters were inputs into the VHDL generator was of paramount importance for LANL's particular situation.

3.3.2 Floating-point cascades

The first step after identifying a base algorithm is to implement these building blocks in floating-point MATLAB to verify that the desired results are achieved. Writing a simple MATLAB module to mimic the radix-2 step and writing a MATLAB function to generate the

twiddle factors needed between cascaded stages to implement larger FFTs was the starting point for this example. Once these two modules were written, *N*-point FFTs were created through combinations of these building blocks and the resulting outputs were compared with the built-in MATLAB FFT function to verify that the desired result was obtained.

3.3.3 Fixed-point cascades

To migrate towards a VHDL design, a fixed-point version of the radix-2 function was then written in MATLAB. This fixed-point module has input parameters that control the desired input and output data bit-widths. The precision of the twiddle factors used in an individual radix-2 operation is also an input parameter. The data is allowed to grow to maximum possible bit-width throughout the radix-2 module. The outputs are truncated prior to being output from the function. The fixed-point design internal to the radix-2 module is designed such that overflow and underflow are impossible.

The next stage in the development cycle was to create arbitrarily sized FFTs based on the fixed-point radix-2 algorithm and the fixed-point twiddle generation module. These fixed-point FFTs built on cascades of fixed-point radix-2 functions were then compared with the floating-point cascades described above and the built-in MATLAB FFT function to verify that correct performance was being achieved.

3.3.4 Bit-true, parameterized mathematical model

Once the basic FFT structures were broken down, a bit-true model for the entire parameterized design was developed. Our previously gained knowledge of building FFTs of different sizes was now applied to building a model for an arbitrary-point FFT. Routines were developed in MATLAB to calculate the data swapping between each pass as well as the data reordering (bit-reversing) before the first pass. The arbitrary-point model calculates the FFT by looping log2(*N*) times for an *N*-point FFT. Each pass through the loop involves data reordering and a radix-2 operation. All data points are run through the radix-2 module, 2 points at a time. The loop portion of the *N*-point FFT function is shown below in Figure 3. The model was verified by setting the precision to full scale, scaling the output of the model and then comparing the results to those from the MATLAB built-in FFT.

```
pass_size = [in_size, pass_size];
% this creates reordered x indicees
rx = reshape(reorder(N)+1,2,N/2);
rx = rx(:);
for pass = 1:log2(N)
    y = reorderProducts(N,pass)+1;  % indicees into a
    if(pass == 1)
        y = rx;
    end
    for j = 2:2:N
        if(pass == 1)
            [a(j-1) a(j)] = radix2_BF(x(rx(j-1)),
            |rx(j)|),w(tr(j/2,pass)|, in_size,
            twiddle_size, pass_size(2)|);
        else
            [a(y(j-1)| a(y|j|))] = radix2_BF(a(y|j-1|),
            a(y|j|), w(tr(j/2,pass|),
            pass_size(pass), twiddle_size,
            pass_size(pass+1));
        end
    end
end
```

Figure 3

3.4 Writing VHDL and VHDL Generators

The next step of the process is to build the VHDL generators. There are four basic steps to building a VHDL generator. The steps are repeated for each function in the design as well as for the top-level design. First, the function is written in VHDL for a specific set of parameters. A 16-point FFT was chosen for this step. The finished code should be tested against the bit-true mathematical model of the function. Once the code has been verified, the VHDL generator can be written. The VHDL is generated from MATLAB with the use of `fprintf` statements embedded in loops to write text to a `.vhd` file. The generator is then used to produce VHDL modules for several different parameter combinations. Finally, the modules are inspected by hand and tested against the bit-true mathematical models.

A portion of the MATLAB code used to generate the twiddle factors in the top-level FFT function is shown below in figure 4. A snippet of VHDL code that was generated from this function is shown in figure 5.

```
w = twidleGen(N,twiddle_size);  % generate twiddles
% size twiddles to given bit-width
w = round(w*(2^(twiddle_size-1)-1)/2^(twiddle_size-1));
wr = real(w(1:N/2));     % real parts of twiddles
wi = imag(w(1:N/2));     % imaginary parts
wr_ind = find(wr < 0);  % find twiddles < 0
wi_ind = find(wi < 0);
% put negative twiddles in 2's complement form
wr(wr_ind) = wr(wr_ind) + 2^twiddle_size;
wi(wi_ind) = wi(wi_ind) + 2^twiddle_size;
fprintf(opf, ...
    'constant TWIDDLE_SIZE\t\t: integer := %d;\n\n', twiddle_size);
fprintf(opf, '\ntype complex_twid is record\n');
fprintf(opf, '\tre\t: signed(TWIDDLE_SIZE-1 downto 0);\n');
fprintf(opf, '\tim\t: signed(TWIDDLE_SIZE-1 downto 0);\n');
fprintf(opf, 'end record complex_twid;\n');
fprintf(opf, ...
    'type twid_array is array (0 to %d) of complex_twid;\n', N/2-1);
fprintf(opf, 'constant w\t: twid_array := (\n');
for i = 0:(N/2-2)
    % twiddle size is divisible by 4, so use hex
    if(mod(twiddle_size,4) == 0)
        fprintf(opf, '\t(x"%s", x"%s"),\t\t\t-- w(%d)\n', ...
            dec2hex(wr(i+1),twiddle_size/4), ...
            dec2hex(wi(i+1),twiddle_size/4), i);
    else    % twiddle size is not divisible by 4, so use binary
        fprintf(opf, '\t("%s", "%s"),\t\t\t-- w(%d)\n', ...
            dec2bin(wr(i+1),twiddle_size), ...
            dec2bin(wi(i+1),twiddle_size), i);
    end
end
```

Figure 4

```
constant TWIDDLE_SIZE : integer := 8;

type complex_twid is record
    re  : signed(TWIDDLE_SIZE-1 downto 0);
    im  : signed(TWIDDLE_SIZE-1 downto 0);
end record complex_twid;
type twid_array is array (0 to 7) of complex_twid;
constant w  : twid_array := (
    (x"7F", x"00"),      -- w(0)
    (x"75", x"CF"),      -- w(1)
    (x"5A", x"A6"),      -- w(2)
    (x"31", x"8B"),      -- w(3)
    (x"00", x"81"),      -- w(4)
    (x"CF", x"8B"),      -- w(5)
    (x"A6", x"A6"),      -- w(6)
    (x"8B", x"CF")       -- w(7)
    );
```

Figure 5

The radix-2 routine (both MATLAB and VHDL) has parameterization built into the design. As a result, this code never changes based on any of the input parameters. For this reason, a VHDL generation routine was not designed for this function.

Once the procedure detailed above has been completed for all functions, a test bench for the full system is written in VHDL and used to test the generated code for several parameter combinations. A make file also needs to be written. Code generators for the test bench and the make file can then be developed as well.

Generating the code from MATLAB saved a significant amount of time that would have otherwise had to be spent copying, pasting and editing. Additionally, the automatic code generation cut down on potential typing errors. Furthermore, modifications to the code can be made in one place in the MATLAB generation code rather than in many points in the VHDL. This is a very useful feature when tweaking the code for increased performance.

3.5 Additional Utilities

Automatic VHDL generators become very powerful when coupled with additional utilities. Several utilities were built to complement the FFT generation program. First, a MATLAB function was built to verify designs as well as generate the code. This function generates the VHDL specified by the input parameters, creates a file of input vectors, allows time for Modelsim to run the VHDL code, runs the same data through the MATLAB bit-true model and compares the VHDL and MATLAB outputs for rapid design verification.

Another utility was written to validate the performance of the fixed-point design selected by the FFT generator input parameters. The validation is done by running synthetic data through the MATLAB built-in FFT (full-precision, floating-point) and the bit-true model and analyzing the differences. This tool can be used to determine a set of bit-width that yield minimum acceptable performance and the point of diminishing returns. A minimum of one bit of growth per pass generally yields respectable results for this algorithm. Finally, a function was built to estimate the number of register bits that will be used to implement the design on an FPGA. The result is expressed both as bits and as a percentage of register bits available in a

Virtex 1000 chip. These two functions can be used together to examine the trade-off between design precision and chip resources use very efficiently.

3.6 Final Design Performance in Hardware

Several FFT designs were run through Synplicity and Xilinx Place and Route to get size and space estimates for the Virtex 1000 chip. Sizes up to 128 points can be implemented and run at speeds over 50 MHz. Several designs have been successfully verified in hardware. Each design tested has been verified on the first attempt, and therefore has not needed debugging. An important strength of this and other MBDVE designs is that they are easily partitioned across several chips because they are built from cascaded structures.

4 Conclusion

The MATLAB-based VHDL Design Environment can be a powerful tool for VHDL development for many common algorithmic situations. The techniques described in this paper allow the production of highly readable, highly parameterized code that will allow the quick deployment of a common algorithm into many different application environments. As described through the example of the FFT, the MBVDE provides an environment where a modicum of up front work makes the generation of application specific code much easier later in the development process. Thus, the resulting code is more portable to varying applications, allowing custom implementations without redesigning the VHDL for every single algorithmic situation.

A side benefit of using MBVDE is that it imposes rigor in the development process. It is natural to take the time to tie up loose ends and spend time documenting when it is clear that the code will be reused in the future. This makes the hand off to other developers much easier. In addition, the algorithm being broken down into its basic structures makes visualization and partitioning across chips much easier.

The power of MBVDE resides in the additional utilities that are built into the environment. These utilities include a bit-true parameterized mathematical model, fixed-point design validation, test vector generation, generation of synthesizable VHDL, VHDL design verification, and chip resource use estimation.

5 References

[1] Alan Oppenheim, Ronald Schafer, “Discrete-Time Signal Processing”, Englewwod Cliffs, NJ, Prentice Hall 1989.

[2] Monson Hayes, “Statistical Digital Signal Processing and Modeling”, New York, John Wiley and Sons, Inc. 1996.

[3] E.B. Hogenauer, “An Economical Class of Digital Filters for Decimation and Interpolation”, IEEE Transactions on Acoustics, Speech and Signal Processing, 29(2):155-162, April 1981.

[4] C. Schuler, “Code Generation Tools for Hardware Implementation of FEC Circuits”, Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems, Pafos, Cyprus, September 5, 1999, pp. 221-224.

[5] M. Diepenhorst, M. van Veelen, J.A.G. Nijhuis, L. Spaanenburg, “Automatic Generation of VHDL Code for Neural Applications”, International Joint Conference on Neural Networks, Washington DC, USA, July 10, 1999, pp. 2302-2305.

[6] Josef Dalcolmo, Rudy Lauwereins, Marleen Ade, “Code Generation of Data Dominated DSP Applications for FPGA Targets”, 9th IEEE International Workshop on Rapid System Prototyping, Leuven, Belgium, June 3-5, 1998, pp.162-167.

[7] Daijin Kim, In-Hyun Cho, “FADIS: An Integrated Development Environment for Automatic Design and Implementation of FLC”, Annual Conference of the North American Fuzzy Information Processing Society, Syracuse, NY, USA, September 21-24, 1997, pp. 33-39.